Exploiting Application Dynamism and Cloud Elasticity for Continuous Dataflows

Alok Kumbhare, Yogesh Simmhan and Viktor K. Prasanna
University of Southern California
Los Angeles, California 90089
{kumbhare, simmhan, prasanna}@usc.edu

ABSTRACT

Contemporary continuous dataflow systems use elastic scaling on distributed cloud resources to handle variable data rates and to meet applications' needs while attempting to maximize resource utilization. However, virtualized clouds present an added challenge due to the variability in resource performance – over time and space – thereby impacting the application's QoS. Elastic use of cloud resources and their allocation to continuous dataflow tasks need to adapt to such infrastructure dynamism. In this paper, we develop the concept of "dynamic dataflows" as an extension to continuous dataflows that utilizes *alternate tasks* and allows additional control over the dataflow's cost and QoS. We formalize an optimization problem to perform both deployment and runtime cloud resource management for such dataflows, and define an objective function that allows trade-off between the application's *value* against resource *cost*. We present two novel heuristics, local and global, based on the variable sized bin packing heuristics to solve this NP-hard problem. We evaluate the heuristics against a static allocation policy for a dataflow with different data rate profiles that is simulated using VM performance traces from a private cloud data center. The results show that the heuristics are effective in intelligently utilizing cloud elasticity to mitigate the effect of both input data rate and cloud resource performance variabilities on QoS.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; D.1.3 [**Concurrent Programming**]: Distributed programming; I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic methods

General Terms

Design, Performance, Algorithms

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC'13, November 17-21 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503240

Keywords

Dataflows, clouds, resource management, optimization, data velocity, runtime adaptation

1. INTRODUCTION

Distributed platforms offer a vital solution space to address problems on "Big Data" [9]. Data analytics platforms have expanded beyond scaling for *volume* and *variety* dimensions of data and started focusing on continuous data with high and uneven data rates – the *velocity* dimension. Stream processing frameworks such as Yahoo's S4 [26], Twitter's Storm, IBM InfoSphere Streams [5], and Spark Streaming [47] provide continuous dataflow programming abstractions to build and deploy tasks graphs as distributed applications at scale on commodity clusters and clouds.

While these systems support high input data rates, they fail to adequately consider fluctuations in the data rates that are observed in the real world. Some statically over- or under-provision resources depending on availability, and consequently trade-off between cost of unused resources during low data rate period and the penalty of high processing latencies during the high data rate period. Others track the changes to the incoming data rates and redeploy the application, but this is not done online and can cause loss of messages or high processing latencies during the switch. Cloud computing platforms offer the ability to elastically acquire and release resources, and some recent continuous dataflow frameworks like Esc [32], WSAggregation [14] and others [17, 12] leverage these to perform online scaling of resources based on application load. Few of them also support active migration of processing elements between resources to improve performance and to maximize resource utilization.

Despite these advances, there are two clear gaps in current research on executing continuous dataflows on elastic public clouds: (1) adapting to changing performance of cloud resources, and (2) offering more flexible cost-benefit trade-offs to users running such dataflows on (commercial) clouds.

Cloud infrastructure exhibits performance variability over time and space. The same virtual machine (VM) instance has different resource characteristics over time due to multitenancy and changing workloads in the data center. Different VM instances of the same resource class (e.g. Medium) show different performance due to placement and diversity in commodity hardware [16]. As a result, scheduling strategies that assume deterministic and homogeneous cloud behavior fail to deliver expected Quality of Service (QoS) to the dataflow applications running on clouds.

In this paper, we address these issues by introducing a

novel concept of "dynamic dataflows", where each task in the dataflow consists of one or more alternate implementations with different cost-benefit ratio. At a given time, the system is free to choose any one of these implementations based on the domain policies and execution metrics. This concept of alternate tasks in dataflows provide another dimension of control, in addition to resource elasticity, which can be leveraged to achieve desired execution goals and constraints in distributed streaming applications.

Utilizing the dynamic dataflow paradigm, we further propose a multi-objective optimization problem to balance the execution cost, processing latency and "value" for the dataflow application. We develop a set of deployment and runtime heuristics that tune both the elastic scaling and the selection of alternate tasks to adapt to the fluctuations in the data rate, as well as, to mitigate the effect of performance variability in the cloud infrastructure.

We evaluate the heuristics through simulations that use real performance traces obtained from a medium-sized private cloud deployment and the empirical results show that the cost of execution for a dataflow can be reduced by as much as 15% given various tolerances in the application throughput and application value.

The rest of the paper is organized as follows. We survey and compare related work in section 2. Section 3 describes the continuous dataflow application model with extensions for dynamic dataflows. Further, we discuss the infrastructure model and parameters in Section 4, with focus on commodity clusters and public clouds. Section 5 presents our approach towards dataflow deployment and runtime adaptation using cloud elasticity and alternate tasks. We formalize the optimization problem in Section 6 followed by multivariate deployment and runtime optimization heuristics in Section 7. We offer implementation details of the dataflow execution and cloud infrastructure simulation in Section 8.1 which is evaluated for various scenarios, as presented in Section 8.2. Section 9 concludes the paper with outlook on future work.

2. RELATED WORK

Scientific workflows [42, 4, 22, 35, 3], continuous dataflow systems [8, 1, 25, 28] and similar large scale distributed programming frameworks [46, 29] have garnered a renewed research focus due to the recent explosion in the amount of data, both archived and real time, and the need for large scale data analysis. Our work is closely related to the stream processing and continuous dataflow systems that allow a task-graph based programming model to execute long running continuous applications which process incoming data streams in near-real time. Other related work includes flexible workflows, service oriented architecture and the notion of "alternates" in heterogeneous computing paradigm. We discuss state-of-the-art in each of these research areas in relation to our proposed system.

2.1 Continuous dataflows, and existing auto scaling techniques

Continuous dataflow systems have their root in Data Stream Management Systems, the primary focus of which is to process continuous queries over tuple streams, composed using a set of welldefined operators [2]. This allows operator-specific query optimizations such as operator split and merge to be performed [33]. General purpose continuous dataflow systems such as S4 [26], Storm[1], and Spark[46], on the other hand, allow arbitrary processing elements, making it necessary to find generic auto scaling solutions such as operator scaling and data parallel operations [48]. Several solutions, including one leveraging cloud elasticity to support auto scaling, have been proposed [12]. However, most of these systems [32, 36], only consider data variability as a factor for auto scaling decisions and assume that the underlying infrastructure offers same performance overtime. Our work shows that this assumption does not hold in virtualized cloud systems and proposes heuristics that handle not only the variations in data rates but also changes in the underlying infrastructure performance.

2.2 Workflow and Task Scheduling

Task based scheduling have been extensively studied and various heuristics proposed. These algorithms are broadly classified into list-scheduling algorithms, clustering, and search based algorithms. Of these, list-scheduling algorithms have received the most attention. The goal of these algorithms is to produce a prioritized list of tasks and place each on the processing unit to minimize a certain cost function (or alternatively, maximize a certain utility function). These algorithms are generally defined for a bounded (and fixed) set of tasks and processors. Our global heuristic is similar to these algorithms; however, the list of task instances as well as their priorities are dynamic and based on external factors.

In addition, several specialized algorithms have been proposed for mapping workflows on to cloud resources with the goal of minimizing the makespan as well as the resource cost [44, 43, 30, 20]. Others use a mix of cloud and local cluster resources to balance performance and cost. However, none of these algorithms consider infrastructure variability of the cloud platform as a factor in scheduling.

2.3 Flexible workflows and SOA

Flexible workflows [27, 31, 39] and service selection in Service Oriented Architecture (SOA) [45, 24, 11] allow workflow compositions to be transformed at runtime. This provides a powerful compositional tool to the developer to define business rule based generic workflows that can be specialized at runtime based on the environmental characteristics. The notion of "alternates" we propose is similar to this in that it allows greater flexibility to the developer and a choice of execution at runtime. However, unlike flexible workflows where the decision about task specialization is made exactly once based on certain deterministic parameters, in continuous dataflows, this decision has to be re-evaluated regularly due to their continuous execution model and dynamic nature of the data streams.

2.4 Heterogeneous computing

To exploit a heterogeneous computing environment, an application task may be composed of several subtasks that have different requirements and performance characteristics. Various dynamic and static task matching and scheduling techniques have been proposed for such scenarios [23, 40, 37]. The concept of alternates in dynamic dataflow is similar to these; however, currently, we do not allow heterogeneous computing requirements for these alternates, though they

[1] HTTP://storm-project.net/

may vary in processing requirements. Even with this restriction, the concept of alternates provides a powerful programming abstraction that allows us to switch between them at runtime to maximize the overall utility of the system in response to changing data rates or infrastructure performance.

2.5 Cloud Infrastructure Performance analysis and modelling

Several studies have compared the performance of the virtualized environment against the barebones hardware to show their average performances are within an acceptable tolerance limit of each other. However these studies focused on the average performance characteristics and not on the variations in performance. Recent analysis of public cloud infrastructure [16, 15, 18] demonstrate high fluctuations in various cloud services, including cloud storage, VM startup and shutdown time as well as virtual machines core performance and virtual networking. Reasons for this include multi-tenancy of VMs on the same physical host, placement of VMs in the data center, use of commodity hardware, collocation of faults, and roll out of software patches to the cloud fabric, among others. Our own studies confirm these. On this basis, we develop an abstraction of the IaaS cloud that incorporates infrastructure variability.

3. DYNAMIC DATAFLOW APPLICATION MODEL

Directed acyclic task (DAG) graphs are familiar constructs used to model loosely as well as tightly coupled applications, and have been adopted by the scientific workflow and dataflow communities, among others. We extend this model to formally define dynamic dataflows that operate on continuous data streams and offer task alternatives.

Def. 1 (Continuous Dataflow). *A continuous dataflow G is a quadruple $G = \{P, E, I, O\}$, where $P = \{P_1, P_2, ..., P_n\}$ is the set of* Processing Elements (PE) *and $E = \{(P_i, P_j) \mid P_i, P_j \in P\}$ is a set of dataflow edges such that there is no cycle, where (P_i, P_j) denotes flow of data* messages *from P_i to P_j. $I \neq \varnothing \subset P$ is a set of input PEs where external data messages enter the dataflow continuously with variable rates, and $O \neq \varnothing \subset P$ is a set of output PEs which emit data messages to be consumed by an external entity.*

Each PE represents a long-running user-defined task in the dataflow which executes continuously, accepting and consuming data messages from the incoming edges and producing messages on the outgoing edges. A directed edge between two PEs connects an output port from the source PE to an input port of the sink PE, and represents a flow of messages between the two [7]. A user can select different semantics for the flow of data on the edges, such as sequence, and-split, synchronize, merge, choice, multi-choice, and so on, similar to dataflow patterns in static workflows [38]. Without loss of generality, we make the simplifying assumption of *and-split semantics* for edges originating from the same output port of a PE (i.e., output messages on a port are duplicated on all outgoing edges) and multi-merge [38] semantics for edges terminating at an input port of another PE (i.e., input messages on a port from all incoming edges are interleaved).

We extend continuous dataflows to "Dynamic Dataflows" by incorporating the concept of alternative PEs (or simply, *alternates*). In heterogeneous computing [23] and flexible (single execution) workflows [39], alternates are *selected once* at runtime and thereafter their dataflow path of execution is fixed. We advance this concept to continuous dataflow where alternate selection is an ongoing process. Alternates enable the user to optionally define alternative implementations for a given PE, each of which may possess different performance characteristics, resource needs and quality of service and hence offer different relative trade-offs between cost and QoS. At the same time, the choice of alternates does not impact the correctness of the application. Due to space constraints, a detailed discussion of "alternates" and the programming model is beyond the scope of this paper. We assume that the alternates have been specified using an appropriate programming model and the set of feasible alternates is available to the proposed heuristics. Alternates defined for different PEs operate independently, and hence any combination of them may be selected. It is also valid to replace one alternate with another during runtime.

Def. 2 (Dynamic Dataflows). *A Dynamic Dataflow D is a continuous dataflow where each PE $P_i \in P$ has set of alternate implementations $P_i = \{p_i^1, p_i^2, ...p_i^j \mid j \geq 1\}$.*

We capture the QoS, cost and performance for each alternate of the PE using a set of metrics, viz., the relative value (γ_i^j), the alternate cost (c_i^j) in core-seconds per message, and the selectivity (s_i^j). The *relative value*, $0 < \gamma_i^j \leq 1$, for an alternate p_i^j is defined as $\gamma_i^j = \frac{f(p_i^j)}{\text{Max}_j\{f(p_i^j)\}}$, where $f(p_i^j)$ is a user-defined value function for the alternate (e.g., the F_1 statistical measure for a classification PE). It quantifies the relative benefit for the user of picking this alternate. The *processing cost*, c_i^j, is the time (in seconds) required to process a single message on a "standard" CPU core (see § 4) for the alternate p_i^j . The *selectivity*, s_i^j, is the ratio of number of the output messages produced to the number of input messages consumed by the alternate p_i^j to complete a logical unit of operation. The selectivity of a PE helps determine the downstream data rate in the dataflow.

A dataflow may be initially deployed with a particular configuration of alternates which can then be switched during runtime to meet the application's constraints. To keep the problem tractable, we enforce that these changes are only made at periodic intervals. During a given time interval t, only one of the alternates for a PE P_i is active, given by:

$$A_i^j(t) = \begin{cases} 1 & \text{if } p_i^j \text{ is active during interval } t \text{ for PE } P_i \\ 0 & \text{otherwise} \end{cases}$$

and $\sum_{p_i^j \in P_i} A_i^j(t) = 1$

The value of the PE P_i in time interval t is then given by:

$$\Gamma_i(t) = \sum_{p_i^j \in P_i} \left(\gamma_i^j \cdot A_i^j(t)\right)$$

We obtain the value for the entire dataflow application during time interval t by aggregating values of the individual alternates since value can be perceived as an additive property [19] over the application graph.

Def. 3 (Normalized Application Value). *The normalized application value $0 < \Gamma(t) \leq 1$ for a dynamic dataflow*

D during time interval t is given by $\Gamma(t) = \frac{\sum_{P_i \in P} \Gamma_i(t)}{|P|}$, *where* $|P|$ *is the number of PEs in dynamic dataflow.*

The application's value thus obtained gives an indication of its overall quality from the user's perspective and can be considered as one dimension of QoS. Yet another dimension, particularly in the context of continuous dataflows, is the observed application throughput. However, raw application throughput is not meaningful because it is a function of the incoming data rates during that time interval. We use the *Relative application throughput*, Ω, as a metric of application performance.

As before, the relative application throughput is built up from the relative PE throughputs of the individual alternates, given by $0 < \Omega_i(t) \leq 1$ for a PE P_i during interval t. It is defined as the ratio of current output data rate (absolute throughput) $o_i(t)$ to the maximum achievable output data rate $o_i^{max}(t)$ given by $\Omega_i(t) = \frac{o_i(t)}{o_i^{max}(t)}$. The output data rate for a PE depends on its alternates' selectivity, and bound by its input data rate and the resources available to process the inputs. The maximum output throughput is achieved when there are enough resources for P_i to process all incoming data messages within the interval t, given by $o_i^{max} = i_i \cdot s_i^j$, where i_i is the input date rate, and s_i^j is the selectivity of its alternate (omitting the time intervals for brevity). The actual output data rate is thus given by $o_i = \frac{min\left(i_i \Delta t, \frac{\phi_i \Delta t}{c_i^j}\right) \cdot s_i^j}{\Delta t}$, where ϕ_i is its total CPU core allocation given by $\phi_i = \sum_j \pi_j$, (where π_j is the core coefficient of the allocated core and j $\in$ allocated cores for the PE (see § 4), and c_i^j is its latency to process a message on a standard core (with $\pi_j = 1$).

The relative throughput for the application is not additive as it depends on the critical processing path of the PEs. We treat the entire dataflow as a black box consisting of a single PE, and use the accumulative incoming data rate at the input PEs and the accumulative output rate at the output PEs to obtain the relative application throughput.

Def. 4 (Relative Application Throughput). *The relative application throughput* $0 < \Omega(t) \leq 1$ *for dataflow D during time t is given by* $\Omega(t) = \frac{\sum_{P_i \in O} \Omega_i(t)}{|O|}$ *where O is the set of output tasks.*

The dynamic dataflows thus extend the familiar DAG application model with support for continuous data streams and dynamic PE alternates. Application value and relative throughput provide complimentary metrics to assess overall application QoS, and allow for clear trade-offs to be made with the resource costs when optimizing the resource scaling and alternate selection.

4. CLOUD INFRASTRUCTURE MODEL

Our heuristics leverage the elasticity of cloud infrastructure at runtime, and we model the behavior of a virtualized commercial cloud Infrastructure as a Service (IaaS) environment here. The execution framework has access only to the virtualized cloud resources: virtualized CPU cores, virtual disks within a VM, and virtual network connectivity. There is no control over or knowledge of the actual VM placement within the data center and, consequently, the network connection behavior between the VMs. This implies different network profiles for different VM instances even of the same class.

The cloud environment consists of a set of VM resource classes $\mathcal{C} = \{C_1, C_2, ..., C_n\}$ that differ in the number of available CPU cores N, their rated core speed π, and their rated network bandwidth β. We assign atleast one dedicated core to each VM. For simplicity, we ignore memory and disk characteristics in the current model. Since CPU core speeds may vary across VM classes, we define the normalized processing power π_i of a resource class C_i's CPU core as the ratio of its processing power to that of a "standard" VM core, under ideal conditions. Naïvely, this may be the ratio of their CPU core clock speeds (e.g, $\frac{2.2\ GHz}{1.7\ GHz}$), but could also be the result of running application benchmarks on a standard VM and the VM of a particular resource class. Cloud providers such as Amazon also provide resource class ratings in the form of Elastic Compute Units (ECUs) that can be used. The processing requirements of a PE alternate is defined in terms of core-seconds (c) required to process a single message on the standard VM core (whose $\pi = 1$). Hence, the latency of a PE to process a message on a resource of class C_i can be obtained by scaling as $c_i = c/\pi_i$. In addition, each resource class is associated with a fixed hourly usage price ζ_i. We follow a costing model similar to existing cloud providers. The usage of a VM instance is rounded up to the nearest hourly boundary and the user is charged for the entire hour even if it is shut down before the hour ends.

$R(t) = \{r_1, r_2, ..., r_n\}$ is a set of all VMs that have been instantiated till time t. Each instance is described by the tuple $r_i = \langle C_j, t_{start}, t_{off} \rangle$, which is the resource class C_j to which the VM instance belongs, t_{start} is the time at which the instance was created and t_{off} is the time at which the instance was turned off. t_{off} is set to ∞ for an active VM. Total accumulated cost for the instance r_i at time t is then calculated as $\mu_i[t] = \lceil min(t_{off}, t) - t_{start})/60 \rceil \times \zeta_j$, where $min(t_{off}, t) - t_{start})$ is the duration for which the instance has been active.

Various studies, including our own, have shown that the performance of cloud VM instances is volatile, over time and across instances of the same class, including its processing power and network bandwidth [16, 18, 15]. To gage the current behavior of the virtualized cloud resource, we presume a monitoring framework that periodically and nonintrusively probes the performance of and faults in the VMs and their network connectivity using standard benchmarks.

The normalized processing power of a VM instance r_i's CPU core at time t thus monitored is given by $\pi_i(t)$, and the network latency and bandwidth between pairs of active VM instances r_i and r_j are $\lambda_{i \times j}(t)$ and $\beta_{i \times j}(t)$ respectively.

Hence, the processing latency for a PE alternate p_i^j at time t is a function of the current normalized processing power for the set of VM instances it is allocated at that time. Similarly, the bandwidth between two PE instances is the current bandwidth between the VM resources on which they are deployed. We assume in-memory message transfer if two PEs are collocated in the same VM instance, i.e., $\lambda_{i \times i} \to 0$ and $\beta_{i \times i} \to \infty$. In addition, during the deployment stage, we assume that the network bandwidth between two VMs is equal to the "rated values". However, during the runtime adaptation, we use the actual bandwidth, reported by the monitoring framework, which may change due to factors including collocation of VMs and data center traffic.

5. DYNAMIC DATAFLOW DEPLOYMENT & ADAPTATION

We outline here the overall approach to deployment and runtime adaptation of the dynamic dataflow on elastic cloud resources. The specific heuristics themselves are discussed in subsequent sections.

Fig. 1(a) shows a simple abstract dataflow with four PEs. E_1 and E_4 are the input and output PEs with just one alternate each, while PEs E_2 and E_3 have of two alternates each. Messages emitted on E_1's output port are duplicated to both E_2 and E_3 while E_4 interleaves the messages from the task parallel operations performed by E_2 and E_3. When this abstract dataflow is submitted for execution, the decisions on alternate selection and VM instance acquisition and allocation fall into two phases: at deployment time and at runtime. Deployment time heuristics (§ 7) determine the initial selection of the alternate for each PE, and the resources required by them based on estimated initial message data rates. Fig. 1(b) shows a sample concrete dataflow where the alternates have been selected, picking e_2^2 and e_3^1 for PEs E_2 and E_3, and the initial VM resources estimated for the four PEs (Fig. 1(c)).

At runtime, several instances of a PE (alternate) operate in a data parallel manner, with each CPU core allocated to the PE being able to operate on independent messages available on their input port. Incoming messages to a VM instance are buffered in a local queue before execution. The mechanics of these, while beyond the scope of this paper, are based on the $\mathcal{F}\ell o\varepsilon$ dataflow framework discussed elsewhere [34] [2]. PEs can also be allocated CPU cores that span multiple VMs; the logical operation of the dataflow is not affected by this though it does impact the performance of the PE due to the variations in processing power and network connectivity across the VMs. We assume incoming messages are load-balanced across the CPU cores, and releasing a VM instance migrates pending input messages in its buffer to remaining VMs hosting this PE (with network cost paid for the transfer). This also means that the PEs are stateless across messages, or can share state only between the instances running on the same VM. This allows us to allocate and de-allocate CPU cores on the same or different VMs rapidly without impacting the dataflow consistency. It also allows us to switch between alternates for a PE.

Based on the deployment time heuristics, VMs of particular resource classes are instantiated and the alternates of the dataflow initiated on them, following which the dataflow execution starts. This also initiates the monitoring probes on the VM instances to record their runtime performance, and measures the message data rates for the alternates. Periodically, we can perform runtime adaptation to respond to variability in incoming data rates and resource behavior using one of several controls that are available. These active decisions are based on runtime heuristics (§ 7) that can decide to pick a different alternate for a PE, change the resources allocated to an alternate within a VM (scale up/down) or on a new VM (scale out/in), or even decide to migrate an alternate from one VM instance to another. The VM acquisition and release are also tied to these decisions as it determines the actual cost paid to the cloud service provider.

This deployment and runtime adaptation model attempts to strike a balance between *simplicity* (e.g., statelessness and isolating PE instance on separate cores to minimize effect on each other), *reality of clouds* (e.g., costing at hour boundaries, network cost to migrate message buffers), and *user flexibility* (e.g., dynamic PEs, runtime VM and dataflow monitoring). This also allows us to formally define a meaningful yet tractable optimization problem for the deployment and runtime adaptation strategies that we introduce next.

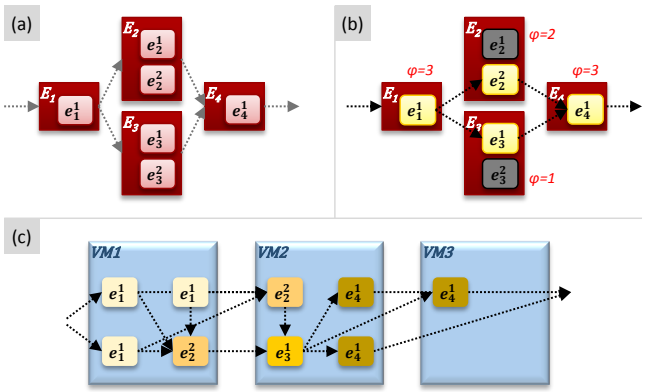

Figure 1: (a) An abstract dynamic dataflow application with possible alternates for PEs, (b) A concrete dynamic dataflow with alternates selected from possible combinations, and the initial resource requirements determined (c) A deployment of the concrete dataflow on VMs

6. PROBLEM FORMULATION

We formulate the optimization problem as a constrained utility maximization problem. The constraints to the function are defined in terms of the expected relative application throughput (Ω) (i.e., the throughput, on average, should not deviate from this requirement at the end of the optimization period). Similarly the utility is defined as a function of the normalized application value (Γ), the maximal cost they are willing to pay for the cloud resources, and the actual cost paid for the resources while achieving the constraint. The goal of the optimization problem is to maximize the utility (objective function) while minimizing the deviation from the defined constraint.

We define an optimization period T for which execution constraints are specified by the user and the utility needs to be maximized. This optimization period is divided into time intervals $T = \{t_0, t_1, ..., t_n\}$. While the period between the intervals could be of variable length, for the purpose of our theoretical analysis we assume that they are of equal length i.e., $t_{i+1} - t_i$ is a constant. The initial deployment decisions are made prior to t_0 and the the runtime decisions made at the beginning of each time interval.

At time t_0, an abstract dynamic dataflow D, along with $\mathcal{I}(t_0) = \{i_j(t_0)\}$, the estimated input data rates at each input PE $P_j \in I$ is given. During each subsequent time interval t, based on the monitoring during the previous time interval, we are also given the *observed* input data rates, $\mathcal{I}(t) = \{i_j(t)\}$; the set of active VM resource instances, $R(t) = \{r_1, r_2, ..., r_m\}$; the normalized processing power per

[2] $\mathcal{F}\ell o\varepsilon$ is available for download at https://github.com/usc-cloud/floe

core for each VM instance r_j, $\pi(t) = \{\pi_j(t)\}$; the network latency and bandwidth between pairs of active VM instances $r_i, r_j \in R(t)$, $\lambda(t) = \{\lambda_{i\times j}(t)\}$ and $\beta(t) = \{\beta_{i\times j}(t)\}$.

At any time interval t, we can calculate the *relative application throughput* ($\Omega(t)$) observed at time t using Def. 4, the *normalized application value* ($\Gamma(t)$) using Def. 3, and the cumulative monetary cost ($\mu[t]$) till time t.

The *average* relative application throughput over the optimization period T is given by $\Omega = \frac{\sum_{t\in T}\Omega(t)}{|T|}$, the *average* application value over period T is $\Gamma = \frac{\sum_{t\in T}\Gamma(t)}{|T|}$, and the *total* resource cost over period T is $\mu = \mu[t_n]$, where t_n is the last time interval in T.

We define the combined objective function which is to be maximized over the optimization period as a profit:

$$\Theta = \Gamma - \sigma \cdot \mu$$

where σ is a user-defined equivalence factor between cost and value given by

$$\sigma = \frac{MaxApplicationValue - MinApplicationValue}{AcceptableCostatMaxVal - AcceptableCostatMinVal}$$

Here, the max and min application values can be calculated from the alternates in the dataflow by picking the alternates with the best and worst values for each PE, while the user provides the acceptable costs to achieve these respective extremes. σ captures an intuitive linear function of the user's pricing expectation, and any cost below this linear function serves as a profit.

The goal is to maximize the objective function Θ while satisfying the constraint that the average relative throughput Ω over the optimization period meets a user-defined threshold $\Omega \geq \widehat{\Omega}$.

Θ can be maximized by choosing appropriate values for the following control parameters at each interval t in the optimization period: $A_i^j(t)$, which gives the active alternate j for the PE P_i; $N(t) = \{N^j(t)\}$, the number of VM instances in $R(t)$ belonging to resource class C_j; $\phi(t) = \{\phi_j(t)\}$ ϕ_t^j, the number of CPU cores available to P_j for data parallel execution; and $M(t) = \{M_{j\times k}(t)\}$, the mapping of a PE instance P_j to the actual resource instance r_k.

7. DEPLOYMENT AND ADAPTATION HEURISTICS

Optimally solving the objective function Θ with the Ω throughput constraint is *NP-Hard*. The proof is outside the scope of this paper, but we present an intuition here. $\Theta = \Gamma - \sigma \cdot \mu$ can be simplified by fixing either Γ or μ and optimizing the other. When the application value Γ is fixed, the problem then reduces to minimizing the execution cost μ of the dataflow, i.e., give the incoming data rate and fixed alternates, place PEs onto the VM instances of different resource classes such that the total cost of resources used is minimized while satisfying the $\Omega \leq \widehat{\Omega}$ constraint. This is similar to the *Variable-sized Bin Packing* problem [10], where given a set of objects and an infinite supply of different sized bins, the objective is to minimize the total space used to pack all the objects in the bins. This has been proven as an NP-Complete problem, and our problem is more complex than this reduced version. We thus propose heuristics to provide an approximate solution to the objective function.

Algorithm 1 Heuristic Algorithm for Initial Deployment

```
1: procedure InitialDeployment(Dataflow D)
                                          ▷ Alternate Selection Stage
2:     for Processing Element P ∈ D do
3:         for Alternate A ∈ P do
4:             μ ← GetCostOfAlternate(D, A, P)
5:             γ ← A.Value
6:             if γ/μ ≥ best then
7:                 best ← γ/μ
8:                 selected ← A
9:             end if
10:        end for
11:    end for
                                          ▷ Resource Allocation Stage
12:    VMClasses ← SortedByDecreasingSize()  ▷ size = N ×π
13:    while Ω ≤ Ω̂ do
14:        if (lastVM.isAvailable = false) then
15:            lastVM ← InitializeVM(VMClasses.First)
16:        end if
17:        P ← GetNextPE()
18:        CurrentAlloc ← AllocationOneCore(P, lastVM)
19:        Ω ← GetEstimatedThruput(D, CurrentAlloc)
20:    end while
21:    for Processing Element P ∈ D do
22:        if (IsOverProvisioned(P)) then
23:            RepackPE(P)
24:        end if
25:    end for
26:    RepackFreeVMs()      ▷ Repack PEs in VMs with free cores to smaller VMs
27: end procedure
```

We develop several heuristics to find an approximate solution to the optimization problem in near real time. While techniques such as integer programming and branch-and-bound have been used to optimally solve some NP-hard problems [41], such tractability does not adequately translate to low latency solutions that are critical to continuous adaptation decisions. The dynamic nature of the application and the infrastructure as well as the tightly-bound decision making interval means that fast heuristics are better suited than slow optimal solutions that may in any case become stale.

We distinguish between the initial deployment strategy and the runtime adaptation strategy. The initial deployment is based on the *estimated* data rate and the *expected* VM performance. At runtime, both of these may vary as observed by the monitoring framework, and the adaptation strategy changes the alternate selection and resource allocation. We propose an initial *local heuristic* that uses information available locally for a particular PE for decisions, and refine this to a *global heuristic* that also considers the impact on downstream PEs in the dataflow.

7.1 Initial Deployment Heuristics

The initial deployment algorithm (Alg. 1) is divided into two stages: alternate selection (lines 2-11) and resource allocation (lines 12-26).

The alternate selection stage ranks each alternate for a PE based on its value to cost ratio (line 4) and chooses the one with the highest ratio. Since we do not know the actual cost for the alternates until resource allocation, the local and global heuristics use cost functions for GetCostOfAlternate (Table 1). The local strategy calculates an alternate's cost based only on its processing requirement. The global

Function	Local Strategy	Global Strategy
GetCostOfAlternate	$A.cost$	$A.cost + S_i \times \sum successor.cost$

GetNextPE	**if** All PEs assigned **then**
	  return PE with lowest Ω_t^j ▷ bottleneck
	else
	  return Next PE in BFS
	end if	

RepackPE	N/A	Move PE to smallest VM big enough for required core-secs
Repack FreeVMs	N/A	Iterative Repacking [21]

Table 1: Functions used in Initial Deployment Strategies

strategy calculates the cost of the alternate as the sum of its processing needs and that of its downstream PEs – intuitively, if an upstream PE has more resources allocated, its output message rate increases and this has a cascading impact on the input rate of the succeeding PEs. A higher selectivity will have a higher impact on the successors since they will produce more output messages. This cost is calculated using a dynamic programming algorithm by traversing the dataflow graph in reverse BFS order rooted at the output PEs.

The resource selection stage follows a procedure similar to the *variable sized bin packing (VBP)* problem. For the initial deployment, in the absence of running VM instances, we assume that each instance from a resource class would behave ideally as per its rated performance. A generic VBP heuristic algorithm picks objects (PEs) in a certain order and puts each in the largest bin (VM instance), creating a new bin if required [13]. Based on some criterion, it chooses to "repack" the objects from the largest bin to smaller bins to avoid wasted space. Both, the order in which bins are chosen and the object selection for repacking affects the quality of the heuristic.

Both the local and global heuristics choose one PE (object) at a time using GetNextPE (line 17) and allocate it to the largest VM resource class, either available or newly instantiated (line 18). While the local strategy does not perform any repacking, the global strategy "repacks" individual PEs (line 23) and all PEs in a VM (line 26) using RepackPE and RepackFreeVMs (Table 1).

The intuition behind the GetNextPE function is to choose PEs in an order that not only reduces the spare capacity on a VM but also limits the message transfer latency between the PEs by collocating neighboring PEs in the dataflow within the same VM. We order the PEs based on a forward BFS traversal rooted at the input PEs, and allocate them resources in that order to increase the probability of collocating neighboring PEs. However, note that the CPU cores required for the individual PEs is not known in advance since the resource requirements depend on the current load which in turn depends on the resource requirements of the preceding PE. Hence, after assigning at least one CPU core to each PE (IncrementAllocation), the deployment algorithm chooses PEs in the order of largest bottlenecks in the dataflow, i.e., lowest relative PE throughputs (Ω). This ensures that PEs needing more resource are chosen first for allocation. This in turn may increase the input rate (and processing load) on the successive PEs, making them the bottlenecks. As a result, we take an iterative approach that incrementally allocates CPU cores to PEs until the throughput constraint is met. Since the resource allocation only impacts downstream PEs, this algorithm is bound to converge. We leave a theoretical proof to future work. Both the local and global strategies use this method to select the next PE for allocation.

The global strategy further performs two levels of repacking. After a solution is obtained using VM instances from just the largest resource class, it first moves the last instance for all the over-provisioned PEs to the smallest resource class large enough to accommodate that instance (best fit, using RepackPE). This may free up spare capacity on the initialized VMs. Hence, we again repack all the VM instances to minimize the wasted cores. Finally, an iterative repacking algorithm [21] is used to allocate cores for all the instances remaining on the last empty VM of the largest size (RepackVMs). During repacking we may sacrifice instance collocation in favor of reduced resource cost. The evaluation section however shows that this is an acceptable trade-off towards maximizing the objective function.

7.2 Runtime Adaptation Heuristics

The runtime adaptation kicks in after the deployed application has executed for a single time interval. Alg. 2 extends from the initial deployment algorithm but in addition considers the current state of the dataflow and cloud resources, available through monitoring, in adapting the alternate and resource selection. This gives a more accurate estimate of data rates (t_{i+1} is similar to t_i), and hence the resource requirements and the cost. As before, the algorithm is divided into two stages: alternate selection and resource allocation. However, unlike earlier, we do not run both the stages at the same time interval, but instead the former is run every m intervals and the latter is run every n intervals. We do so to keep a balance between application value which is decided by the alternate selection stage and the resource cost decided by the resource allocation stage.

During the alternate selection stage, we assume that the resource behavior will remain static for the next time interval. Hence, given the current data rate and resource performance, we calculate the resources needed for each alternate of a PE. As before, we use local and global variations of the strategy (line 5). We then create a list of "feasible" alternates for a given PE (lines 6-13), based on whether the current relative throughput is less or more than the expected throughput by a threshold. Finally, we sort the feasible alternates in the decreasing order of the value to cost ratio and select the first alternate which can be accommodated under the given available resources. The outcome of this phase is that the system either increases or decreases the overall value based on whether it is overprovisioned or underprovisioned respectively.

The resource re-deployment procedure is used to allocate or de-allocate resources as required to maintain the required relative throughput and to minimize the overall cost. If the average relative throughput thus far is less than $\widehat{\Omega}$, the algorithm proceeds similar to the initial deployment algorithm. It incrementally allocates additional resources to the bot-

Algorithm 2 Heuristic Algorithm for Runtime Adaptation

```
1: procedure AlternateReDeploy(DataflowD, Ω_t)  ▷ Ω_t is the observed relative throughput
2:     for PEP ∈ D do                            ▷ Alternate selection phase
3:         actual ← getAvailableResources()      ▷ Gets the current available resources
4:         for AlternateA ∈ P do
5:             actual ← actualResourceRequirements(A)
6:             if Ω_t ≤ Ω̂ − ϵ then
7:                 if actual ≤ activeAlternate.c then
8:                     feasible.add(A)
9:                 end if
10:            else if Ω_t ≥ Ω̂ + ϵ then
11:                if actual ≥ activeAlternate.c then
12:                    feasible.add(A)
13:                end if
14:            end if
15:        end for
16:        Sort(feasible)                        ▷ decreasing order of value/cost
17:        for feasible alternate A do
18:            if actual < available then
19:                SwitchAlternate(A)
20:                done
21:            end if
22:        end for
23:    end for
24: end procedure
1: procedure ResourceReDeploy(X)
2:     VMClasses ← SortedByDecreasingSize()      ▷ size = N ×π
3:     if Ω_t ≤ Ω̂ then
4:         Same procedure as initial deployment
5:     else if Ω_t ≥ Ω̂ then
6:         while Ω ≥ Ω̂ do
7:             PE ← over − provisioned
8:             PEInstance ← selectInstanceToKill()
9:             Ω ← ExpectedOmega(D, CurrentAllocations)
10:        end while
11:        repackFreeVMs()                       ▷ Repack PEs in the VMs with free cores onto smaller VMs to reduce wasted space
12:    end if
13: end procedure
```

tlenecks observed in the system and repacks the VMs if required. However, if $\Omega \leq \widehat{\Omega}$, the system has two decisions to make: first, which PE needs to be scaled in and second, which instance of the PE to be removed (thus freeing the resource core). As before, different heuristics use different strategies to make this decision.

Both the local and the global strategy picks the over-provisioned PE which has the largest Ω_t during the previous interval. The local strategy then randomly picks an instance of the PE to be removed, while the global strategy picks a PE instance which is on a VM instance with the lowest "value", as determined by the following function:

$VM_value = TimeLeftInCurrentCostCycle \times \frac{FreeCores}{TotalCores} \times (1 - \frac{n}{N}) \times \frac{CoreCount \times \pi}{cost}$, where n is the number of PE instances for a given PE on this VM, and N is the total number of instances for that PE across VMs.

The VM value, as determined by the above function, weighs those VMs with more time left in the cycle higher than those with less time. Similarly, it positively weighs VMs with higher free core percentages. Using this metric to pick the PE instance ensures that the VMs with lower value are freed up and shutdown at the end of their cost cycle to effectively reduce the resource cost.

8. EVALUATION

We evaluate the proposed heuristics through extensive simulations using an IaaS cloud simulator that uses performance traces from a real deployment on a medium sized private cloud infrastructure. Our *IaaSSimulator* is similar, in principle, to CloudSim [6] in that it allows users to test the expected performance and cost characteristics of their applications on a cloud environment without the (prohibitive) cost and effort of an actual deployment. However, unlike CloudSim, we provide an abstraction at the IaaS layer and hide the details of the underlying data center architecture. In addition, to mimic the cloud behavior – especially the observed performance variations – IaaSSimulator supports infrastructure dynamism and also allows replaying performance traces gathered from real cloud systems to accurately simulate the infrastructure characteristics [3].

8.1 Experiments

The experimental setup consists of a small abstract dynamic dataflow with several alternates as shown in Fig. 1. Even though we use a small abstract dataflow, it is scaled up to 10's of alternates and 100's of VMs to accommodate high and variable data rates that demonstrates scalability of the proposed heuristics. We use the same virtual machine instance types as provided by the AWS cloud provider with similar performance ratings and on-demand pricing per hour. For simplicity, during initial deployment, we assume an average network bandwidth between VMs of any types to be 100 Mbps.

We use several profiles to simulate both the data and infrastructure characteristics. To simulate typical streaming data characteristics in continuous dataflows, we use three profiles, viz., constant data rate, periodic waves, and random walk around a mean. We perform several experiments at different data rates ranging from 2 msgs/sec to 50 msgs/sec with a size of ∼100 KBytes/msg.

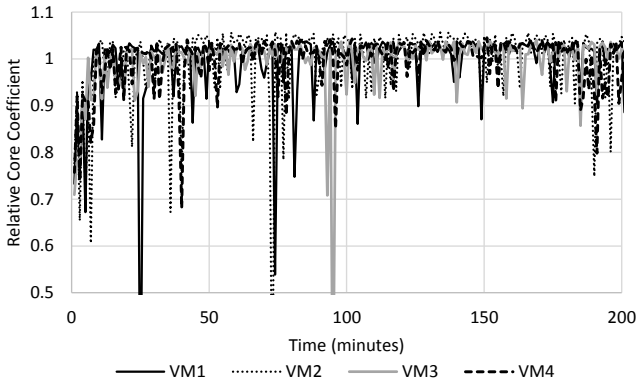

Figure 2: Variations in VM CPU performance in a private IaaS cloud

We simulate infrastructure variability by replaying the performance traces obtained from running periodic micro-benchmarks on 20 VMs in the FutureGrid Eucalyptus 2.0 private cloud for 26 days. Fig. 2 plots a representative sample of metrics from four VMs, showing the CPU performance

[3]IaaSSimulator is available for download at https://github.com/usc-cloud/IaaSSimulator

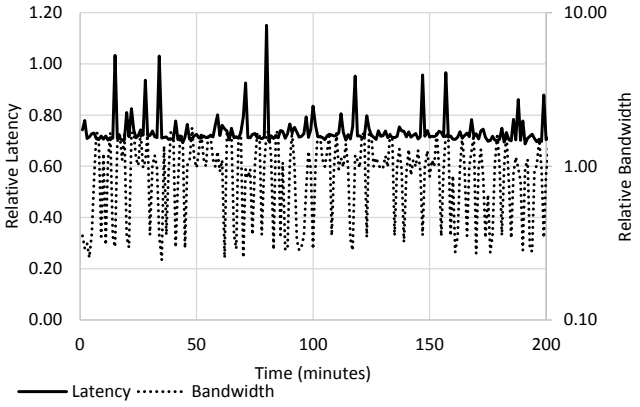

Figure 3: Variations in network performance between a pair of VMs in a private IaaS cloud

variability for those VMs over a four day period and the relative deviation of CPU performance from its mean. Similarly, Fig. 3 shows variations in network latency and bandwidth availability between a pair of VMs over the same time period. Our experiments on the FutureGrid private cloud conform with other such studies [16] that illustrate high variations in the performance of IaaS clouds. For individual experimental runs, we assign a random time period from the traces for each active VM to replay. We then multiply that coefficient with the rated performance of the active VM to obtain its instantaneous runtime performance.

To evaluate the efficacy of the proposed heuristics, we compare them against several other implementations including a static bruteforce optimal deployment for small graphs (that assumes no variations) as well as a static deployment using the local and the global heuristic. To specifically evaluate the benefits of using our notion of dynamic applications, we also compare our heuristics against a simplified version which ignores alternate selection as an optimization decision.

8.2 Results

To judge an algorithm to be better than the other, we first compare if both the algorithms were able to satisfy the user defined constraint on the relative application throughput (i.e. if $\Omega \geq \hat{\Omega} - \epsilon$. Note that this is just a necessary constraint, and high values of Ω beyond $\hat{\Omega}$ does not signify a better algorithm. For that, we use the overall Θ value, $\Theta = \Gamma - \sigma \times TotalCost$, as the second level comparator. Similarly, an algorithm that gives a higher Θ value is in itself insufficient if it does not meet the Ω constraint. For all the experimental runs, we define $\hat{\Omega} = 0.7$ and to calculate σ, we use expected cost at maximum application value to be $\frac{4 \times T \times (DataRate)}{2}$, i.e. for the given small workflow we assign a value of \$4/hour for execution at 2 msgs/sec and then scale it linearly upto \$100/hour for 50 msgs/sec. We empirically arrive at these numbers through observing the actual cost for executing the workflow using a static deployment model.

We first discuss the effect of different types of variability on the relative throughput for static deployments. We compare the static brute-force solution against the static deployments using the local and global heuristics. No continuous monitoring or actions are performed. Fig. 4 shows relative throughput (Ω) on the Y axis for different scenarios at a fixed data rate of 5 msgs/sec. With no (data or infrastructure) variability, the bruteforce algorithm performs the best (i.e. it always satisfies the Ω constraint and gives the highest Θ value), followed by the local heuristic and then the global heuristic. For small data rates, the local heuristic performs slightly better in terms of Ω because global heuristic involves repacking the empty virtual machines which sacrifices PE collocation. However, as we see later, this issue is mitigated by the continuous version of the global heuristic. As we introduce variability in data, infrastructure or both, the performance of even the bruteforce static deployment decreases drastically and none of the algorithms are able to satisfy the Ω constraint, though the Θ value remains the same. This proves the need for continuous redeployment strategies. Besides the brute-force algorithm which takes prohibitively long to find a solution for higher data rates, the performance of both the static deployment algorithms (i.e., local static and global static) decreases as the data rate increases, even with no variations (Fig. 5). This further accentuates the need for continuous monitoring and redeployment.

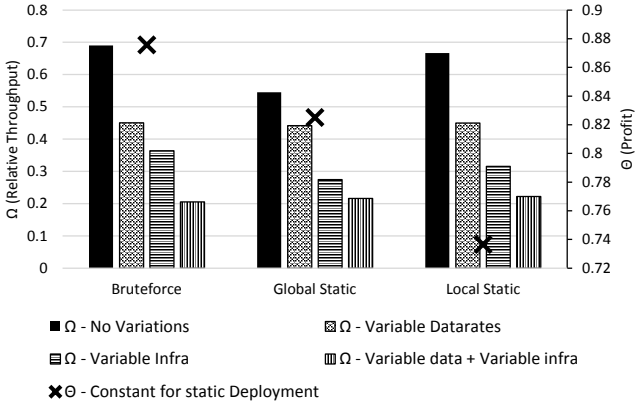

Figure 4: Effect of infrastructure and/or data rate variability on relative throughput, for static deployments

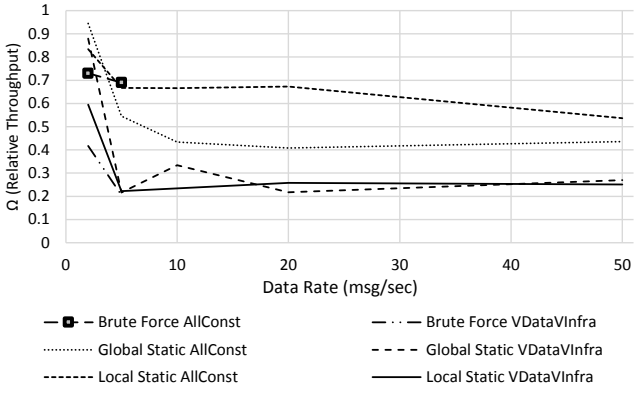

Figure 5: Effect of data rates on relative throughput, for static deployments

Next, we compare the local heuristic against the global

heuristic which considers the cost of all downstream PEs to make the redeployment decision. The obvious advantage of the former is that it does not require centralized monitoring information and can make the decisions locally. However, it suffers from a severe overhead since local decisions may lead to sub-optimal configurations whose impact is realized only after it has cascaded and observed on downstream PEs. This can cause local actions to be reversal which, however, can still lead to a higher cost, say, if the action had initialized a new VM which is priced by the hour. We study these heuristics under different conditions.

Fig. 6 compares the two heuristics under the scenario where the infrastructure performance remains fairly constant but there is variation in the incoming data rates. Such a scenario might occur in a local cluster or an exclusive private cloud where the prospect of multi-tenancy is limited. We observe that both the local and the global heuristics meet the Ω constraint (within an $\epsilon \leq 0.05$ threshold) for the range of data rates. In addition, we see that the global heuristic performs better with respect to the Θ value for data rates higher than 10 msgs/sec. This is expected. The global heuristic makes an informed decision by considering the effect of changes in resources (which changes the output data rate) or alternates (which changes the selectivity, and hence also the output data rate) on the cost of the downstream PEs. Hence, it avoids the penalty due to a reversal of actions. For lower data rates, however, the global heuristic over estimates the effect of the data rate on down stream PEs and avoids re-deployment to reduce the cost or increase the application value. This sometimes leads to lower Θ values, and is more pronounced at lower data rates.

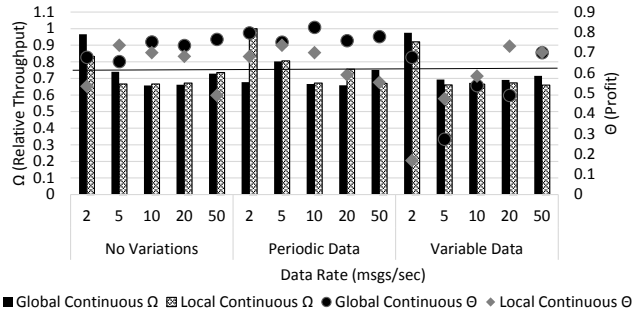

Figure 6: Performance of Local vs. Global Heuristic for stable infrastructure and variation in data rate

Fig. 7 compares the two heuristics at the other extreme, where the data rate remains constant but there are variations in the infrastructure performance. Here, as well, we observe both the local and the global heuristics meeting the Ω constraint (within an $\epsilon \leq 0.05$ threshold) for the range of data rates. As before, we also see that the global heuristic leads to higher Θ value for high data rates, while the local heuristic fairs well for smaller data rates ($\leq 5msg/s$). Further Fig. 8, compares the two under both data and infrastructure variability. We see similar trends in this graph too, with both the heuristics satisfying the Ω constraint and the global heuristic significantly out-performing the local heuristic at higher data rates (leading to a $\sim$60% reduction in the overall dollar cost). However, the local heuristic does continue to perform much better for small data rates

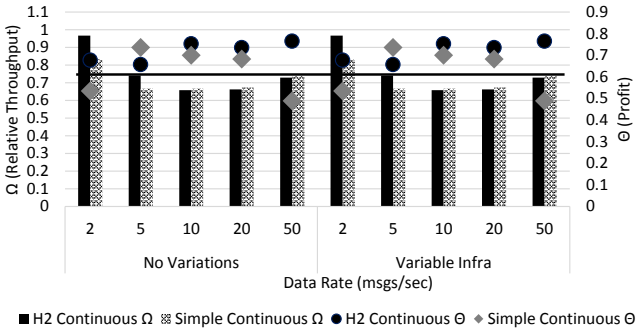

Figure 7: Performance of Local vs. Global Heuristic for stable data rate and variation in infrastructure

(2 msg/sec).

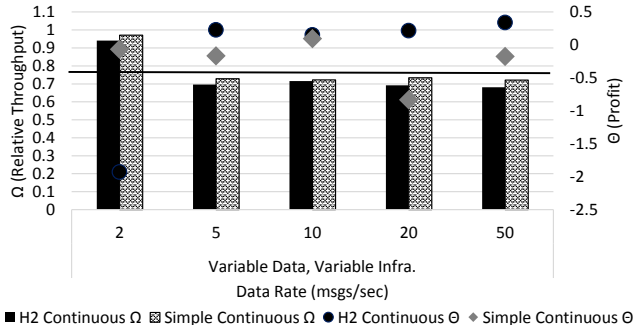

Figure 8: Performance of Local vs. Global Heuristic for variations in both data rates and infrastructure

In summary, so far we have seen that the static algorithms (including static brute-force optimal) perform poorly in the presence of any form of (data or infrastructure) variability. While the global heuristic performs better than the local heuristic in presence of either or both forms of variability at high data rates, the local heuristic does better with low data rates. Both the heuristics take advantage of the alternates provided by the dynamic tasks to increase the overall Θ value. Finally, we characterize the unique advantage offered by the alternates in the dynamic applications. We compare variations of the proposed heuristics while enabling or disabling application dynamism (i.e. the alternate selection stage) and hence compare the selective advantage achieved through application dynamism.

Fig. 9 compares the dollar amount spent by various heuristics over a period of 10 hours for various data rates. The heuristics we compare are global, global without application dynamism, local, and local without application dynamism. As seen before, global performs the best with minimum dollar cost for higher data rates and the local heuristic does so for smaller data rates. We also see that the global heuristic without application dynamism costs more than global in every single case. On an average global requires about 15% less dollar amount than the global without application dynamism. The global heuristic out-performs the local heuristic without application dynamism and allows for a savings of upto 70%.

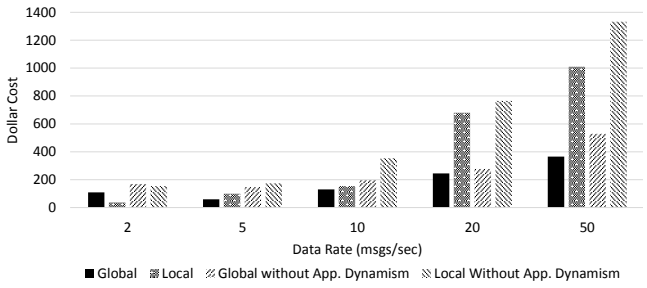

Figure 9: Dollar cost benefit of application dynamism with continuous re-deployment

9. CONCLUSION

In this paper, we have empirically demonstrated the need for online monitoring and re-deployment of continuous dataflow applications to meet their QoS constraints in the presence of either data or infrastructure variability or both. We further introduced the notion of dynamic dataflows with support for alternate implementations for the tasks in the dataflow. This not only gives user the flexibility in terms of application composition, but also gives an additional point of control which can be leveraged to meet the application constraints while maximizing the application value. Our experimental results show that the continuous re-deployment heuristic which makes use of application dynamism can reduce the execution cost by up to 15% on clouds while also meeting the QoS constraints.

As future work, we propose to extend the concept of dynamic tasks to dynamic paths. This will further allow for alternate implementations at coarser granularities, such as a subset of the application graph, and provide end user with more sophisticated and intuitive controls. In addition, we also plan to investigate the application of dynamic tasks to support enhanced fault tolerance and recovery mechanisms in continuous dataflow.

10. ACKNOWLEDGMENTS

This work was supported by grants from the DARPA XDATA and NSF CiC programs (CCF-1048311). We thank FutureGrid for resources provided under NSF Award 0910812.

11. REFERENCES

[1] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. Spc: A distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*. ACM, 2006.

[2] S. Babu and J. Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[3] R. Barga, J. Jackson, N. Araujo, D. Guo, N. Gautam, and Y. Simmhan. The trident scientific workflow workbench. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, 2008.

[4] A. Barker and J. Van Hemert. Scientific workflow: a survey and research directions. In *Parallel Processing and Applied Mathematics*, pages 746–753. 2008.

[5] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran. Ibm infosphere streams for scalable, real-time, intelligent transportation services. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010.

[6] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[7] S. Carlsen. Action port model: A mixed paradigm conceptual workflow modeling language. In *Cooperative Information Systems, 1998. Proceedings. 3rd IFCIS International Conference on*, 1998.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003.

[9] L. Douglas. 3d data management: Controlling data volume, velocity, and variety. Technical report, Gartner, 2001.

[10] D. K. Friesen and M. A. Langston. Variable sized bin packing. *SIAM journal on computing*, 15(1):222–230, 1986.

[11] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio. Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *Services Computing, IEEE Transactions on*, 3(3):223–235, 2010.

[12] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. 2012.

[13] M. Haouari and M. Serairi. Heuristics for the variable sized bin-packing problem. *Computers & Operations Research*, 36(10):2877–2884, 2009.

[14] W. Hummer, P. Leitner, and S. Dustdar. Ws-aggregation: distributed aggregation of web services data. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011.

[15] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):931–945, 2011.

[16] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of production cloud services. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*.

[17] A. Ishii and T. Suzumura. Elastic stream computing with clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, july 2011.

[18] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 159–168. IEEE, 2010.

[19] M. C. Jaeger, G. Rojec-Goldmann, and G. Muhl. Qos aggregation for web service composition using

workflow patterns. In *Enterprise distributed object computing conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International*, pages 149–159, 2004.

[20] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. Scientific workflow applications on amazon ec2. In *E-Science Workshops, 2009 5th IEEE International Conference on*, pages 59–66. IEEE, 2009.

[21] J. Kang and S. Park. Algorithms for the variable sized bin packing problem. *European Journal of Operational Research*, 147(2):365–372, 2003.

[22] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[23] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and distributed Computing*, 59(2):107–131, 1999.

[24] E. M. Maximilien and M. P. Singh. A framework and ontology for dynamic web services selection. *Internet Computing, IEEE*, 8(5):84–93, 2004.

[25] P. Neophytou, P. Chrysanthis, and A. Labrinidis. Confluence: Implementation and application design. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2011 7th International Conference on*, oct. 2011.

[26] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177.

[27] G. J. Nutt. The evolution towards flexible workflow systems. *Distributed Systems Engineering*, 3(4), 1996.

[28] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. Rao, V. Sankarasubramanian, S. Seth, et al. Nova: continuous pig/hadoop workflows. In *international conference on Management of data*, 2011.

[29] S. Pallickara, J. Ekanayake, and G. Fox. Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009.

[30] R. Prodan and T. Fahringer. Dynamic scheduling of scientific workflow applications on the grid: a case study. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 687–694. ACM, 2005.

[31] S. W. Sadiq, M. E. Orlowska, and W. Sadiq. Specification and validation of process constraints for flexible workflows. *Information Systems*, 30(5):349–378, 2005.

[32] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar. Esc: Towards an elastic stream computing platform for the cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*.

[33] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 25–36. IEEE, 2003.

[34] Y. Simmhan, A. Kumbhare, and C. Wickramachari. Floe: A dynamic, continuous dataflow framework for elastic clouds. Technical report, USC, 2013. http://ceng.usc.edu/šimmhan/pubs/simmhan-usctr-2013.pdf.

[35] I. J. Taylor, E. Deelman, and D. B. Gannon. Workflows for e-science: scientific workflows for grids.

[36] R. Tolosana-Calasanz, J. At'ngel Bañares, C. Pham, and O. Rana. End-to-end qos on shared clouds for highly dynamic, large-scale sensing data streams. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 904–911. IEEE, 2012.

[37] H. Topcuoglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[38] W. M. van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003.

[39] J. Wainer and F. de Lima Bezerra. Constraint-based flexible workflows. In *Groupware: Design, Implementation, and Use*, pages 151–158. 2003.

[40] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1):8–22, 1997.

[41] G. J. Woeginger. Exact algorithms for np-hard problems: A survey. In *Combinatorial Optimization—Eureka, You Shrink!*, pages 185–207. Springer, 2003.

[42] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *Sigmod Record*, 34(3):44, 2005.

[43] J. Yu and R. Buyya. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Scientific Programming*, 14(3):217–230, 2006.

[44] J. Yu, R. Buyya, and C. K. Tham. Cost-based scheduling of scientific workflow applications on utility grids. In *e-Science and Grid Computing, 2005. First International Conference on*, pages 8–pp. IEEE, 2005.

[45] T. Yu, Y. Zhang, and K.-J. Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web*, 1(1):6, 2007.

[46] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[47] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10, 2012.

[48] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 54–71. Springer, 2006.